

## 13 | 五花八门的算法：不要再手写for循环了

2020-06-04 罗剑锋

罗剑锋的C++实战笔记

[进入课程 >](#)



讲述：Chrono

时长 15:18 大小 14.02M



你好，我是 Chrono。

上节课我提到了计算机界的经典公式“算法 + 数据结构 = 程序”，公式里的“数据结构”就是 C++ 里的容器，容器我们已经学过了，今天就来学习下公式里的“算法”。

虽然算法是 STL（标准库前身）的三大要件之一（容器、算法、迭代器），也是 C++ 标准库里一个非常重要的部分，但它却没有像容器那样被大众广泛接受。

从我观察到的情况来看，很多人都会在代码里普遍应用 vector、set、map，但几乎从不用任何算法，聊起算法这个话题，也是“一问三不知”，这的确是一个比较奇怪的现象。而且，很多语言对算法也不太“上心”。

但是，在 C++ 里，算法的地位非常高，甚至有一个专门的“算法库”。早期，它是泛型编程的示范和应用，而在 C++ 引入 lambda 表达式后，它又成了函数式编程的具体实践，所以，**学习掌握算法能够很好地训练你的编程思维，帮你开辟出面向对象之外的新天地。**

## 认识算法

从纯理论上来说，算法就是一系列定义明确的操作步骤，并且会在有限次运算后得到结果。

计算机科学里有很多种算法，像排序算法、查找算法、遍历算法、加密算法，等等。但是在 C++ 里，算法的含义就要狭窄很多了。

C++ 里的算法，指的是**工作在容器上的一些泛型函数**，会对容器内的元素实施的各种操作。

C++ 标准库目前提供了上百个算法，真的可以说是“五花八门”，涵盖了绝大部分的“日常工作”。比如：

remove, 移除某个特定值；

sort, 快速排序；

binary\_search, 执行二分查找；

make\_heap, 构造一个堆结构；

.....

不过要是“说白了”，算法其实并不神秘，因为所有的算法本质上都是 for 或者 while，通过循环遍历来逐个处理容器里的元素。

比如说 count 算法，它的功能非常简单，就是统计某个元素的出现次数，完全可以用 range-for 来实现同样的功能：

```
1 vector<int> v = {1,3,1,7,5};    // vector容器
2
3 auto n1 = std::count(           // count算法计算元素的数量
4     begin(v), end(v), 1         // begin()、end()获取容器的范围
5 );
```

 复制代码

```

6  int n2 = 0;
7  for(auto x : v) {                // 手写for循环
8      if (x == 1) {                // 判断条件，然后统计
9          n2++;
10     }
11 }
12

```

你可能会问，既然是这样，我们直接写 for 循环不就好了吗，为什么还要调用算法来“多此一举”呢？

在我看来，这应该是一种“境界”，**追求更高层次上的抽象和封装**，也是函数式编程的基本理念。

每个算法都有一个清晰、准确的命名，不需要额外的注释，让人一眼就可以知道操作的意图，而且，算法抽象和封装了反复出现的操作逻辑，也更有利于重用代码，减少手写的错误。

还有更重要的一点：和容器一样，算法是由那些“超级程序员”创造的，它的内部实现肯定要比你随手写出来的循环更高效，而且必然经过了良好的验证测试，绝无 Bug，无论是功能还是性能，都是上乘之作。


如果在以前，你不使用算法还有一个勉强可以说的理由，就是很多算法必须要传入一个函数对象，写起来很麻烦。但是现在，因为有可以“**就地定义函数**”的 lambda 表达式，算法的形式就和普通循环非常接近了，所以刚刚说的也就不再是什么问题了。

用算法加上 lambda 表达式，你就可以初步体验函数式编程的感觉（即函数套函数）：

```

1  auto n = std::count_if(           // count_if算法计算元素的数量
2      begin(v), end(v),             // begin()、end()获取容器的范围
3      [](auto x) {                  // 定义一个lambda表达式
4          return x > 2;              // 判断条件
5      }
6  );                                // 大函数里面套了三个小函数

```

 复制代码

## 认识迭代器

在详细介绍算法之前，还有一个必须要了解的概念，那就是迭代器（iterator），它相当于算法的“手脚”。

虽然刚才我说算法操作容器，但实际上它看到的并不是容器，而是指向起始位置和结束位置的迭代器，算法只能通过迭代器去“**间接**”访问容器以及元素，算法的能力是由迭代器决定的。


这种间接的方式有什么好处呢？

这就是泛型编程的理念，与面向对象正好相反，**分离了数据和操作**。算法可以不关心容器的内部结构，以一致的方式去操作元素，适用范围更广，用起来也更灵活。

当然万事无绝对，这种方式也有弊端。因为算法是通用的，免不了对有的数据结构虽然可行但效率比较低。所以，对于 merge、sort、unique 等一些特别的算法，容器就提供了专门的替代成员函数（相当于特化），这个稍后我会再提一下。


C++ 里的迭代器也有很多种，比如输入迭代器、输出迭代器、双向迭代器、随机访问迭代器，等等，概念解释起来不太容易。不过，你也没有必要把它们搞得太清楚，因为常用的迭代器用法都是差不多的。你可以把它简单地理解为另一种形式的“智能指针”，只是它**强调的是对数据的访问**，而不是生命周期管理。

容器一般都会提供 begin()、end() 成员函数，调用它们就可以得到表示两个端点的迭代器，具体类型最好用 auto 自动推导，不要过分关心：

 复制代码

```
1 vector<int> v = {1,2,3,4,5};    // vector容器
2
3 auto iter1 = v.begin();         // 成员函数获取迭代器，自动类型推导
4 auto iter2 = v.end();
```

不过，我建议你使用更加通用的全局函数 begin()、end()，虽然效果是一样的，但写起来比较方便，看起来也更清楚（另外还有 cbegin()、cend() 函数，返回的是常量迭代器）：

 复制代码

```
1 auto iter3 = std::begin(v);    // 全局函数获取迭代器，自动类型推导
```

迭代器和指针类似，也可以前进和后退，但你不能假设它一定支持 “++” “--” 操作符，最好也要用函数来操作，常用的有这么几个：

`distance()`，计算两个迭代器之间的距离；

`advance()`，前进或者后退 N 步；

`next()/prev()`，计算迭代器前后的某个位置。

你可以参考下面的示例代码快速了解它们的作用：

 复制代码

```
1 array<int, 5> arr = {0,1,2,3,4}; // array静态数组容器
2
3 auto b = begin(arr);           // 全局函数获取迭代器，首端
4 auto e = end(arr);            // 全局函数获取迭代器，末端
5
6 assert(distance(b, e) == 5); // 迭代器的距离
7
8 auto p = next(b);              // 获取“下一个”位置
9 assert(distance(b, p) == 1);   // 迭代器的距离
10 assert(distance(p, b) == -1); // 反向计算迭代器的距离
11
12 advance(p, 2);                // 迭代器前进两个位置，指向元素'3'
13 assert(*p == 3);
14 assert(p == prev(e, 2));      // 是末端迭代器的前两个位置
```

## 最有用的算法

接下来我们就要大量使用各种函数，进入算法的函数式编程领域了。

### 手写循环的替代品

首先，我带你来认识一个最基本的算法 `for_each`，它是手写 `for` 循环的真正替代品。

`for_each` 在逻辑和形式上与 `for` 循环几乎完全相同：

 复制代码

```
1 vector<int> v = {3,5,1,7,10}; // vector容器
```

```

2   for(const auto& x : v) {           // range for循环
3       cout << x << ",";
4   }
5
6   auto print = [](const auto& x)    // 定义一个lambda表达式
7   {
8       cout << x << ",";
9   };
10  for_each(cbegin(v), cend(v), print); // for_each算法
11
12  for_each(                          // for_each算法, 内部定义lambda表达式
13      cbegin(v), cend(v),           // 获取常量迭代器
14      [](const auto& x)             // 匿名lambda表达式
15      {
16          cout << x << ",";
17      }
18  );
19

```

初看上去 `for_each` 算法显得有些累赘，既要指定容器的范围，又要写 `lambda` 表达式，没有 `range-for` 那么简单明了。

对于很简单的 `for` 循环来说，确实是如此，我也不建议你对这么简单的事情用 `for_each` 算法。

但更多的时候，`for` 循环体会做很多事情，会由 `if-else`、`break`、`continue` 等语句组成很复杂的逻辑。而单纯的 `for` 是“无意义”的，你必须去查看注释或者代码，才能知道它到底做了什么，回想一下曾经被巨大的 `for` 循环支配的“恐惧”吧。

`for_each` 算法的价值就体现在这里，它把要做的事情分成了两部分，也就是两个函数：一个**遍历容器元素**，另一个**操纵容器元素**，而且名字的含义更明确，代码也有更好的封装。

我自己是很喜欢用 `for_each` 算法的，我也建议你尽量多用 `for_each` 来替代 `for`，因为它能够促使我们更多地以“函数式编程”来思考，使用 `lambda` 来封装逻辑，得到更干净、更安全的代码。


## 排序算法

`for_each` 是 `for` 的等价替代，还不能完全体现出算法的优越性。但对于“排序”这个计算机科学里的经典问题，你是绝对没有必要自己写 `for` 循环的，必须坚决地选择标准算法。



在求职面试的时候，你也许手写过不少排序算法吧，像选择排序、插入排序、冒泡排序，等等，但标准库里的算法绝对要比你能写出的任何实现都要好。

说到排序，你脑海里跳出的第一个词可能就是 `sort()`，它是经典的快排算法，通常用它准没错。

 复制代码

```
1 auto print = [](const auto& x) // lambda表达式输出元素
2 {
3     cout << x << ", ";
4 };
5
6 std::sort(begin(v), end(v)); // 快速排序
7 for_each(cbegin(v), cend(v), print); // for_each算法
```

不过，排序也有多种不同的应用场景，`sort()` 虽然快，但它是不稳定的，而且是全排所有元素。

很多时候，这样做的成本比较高，比如 TopN、中位数、最大最小值等，我们只关心一部分数据，如果你用 `sort()`，就相当于“杀鸡用牛刀”，是一种浪费。

C++ 为此准备了多种不同的算法，不过它们的名字不叫 `sort`，所以你要认真理解它们的含义。

我来介绍一些常见问题对应的算法：

要求排序后仍然保持元素的相对顺序，应该用 `stable_sort`，它是稳定的；

选出前几名（TopN），应该用 `partial_sort`；

选出前几名，但不要求再排出名次（BestN），应该用 `nth_element`；

中位数（Median）、百分位数（Percentile），还是用 `nth_element`；

按照某种规则把元素划分成两组，用 `partition`；

第一名和最后一名，用 `minmax_element`。

下面的代码使用 `vector` 容器示范了这些算法，注意它们“函数套函数”的形式：

```

1 // top3
2 std::partial_sort(
3     begin(v), next(begin(v), 3), end(v)); // 取前3名
4
5 // best3
6 std::nth_element(
7     begin(v), next(begin(v), 3), end(v)); // 最好的3个
8
9 // Median
10 auto mid_iter = // 中位数的位置
11     next(begin(v), v.size()/2);
12 std::nth_element( begin(v), mid_iter, end(v)); // 排序得到中位数
13 cout << "median is " << *mid_iter << endl;
14
15 // partition
16 auto pos = std::partition( // 找出所有大于9的数
17     begin(v), end(v),
18     [](const auto& x) // 定义一个lambda表达式
19     {
20         return x > 9;
21     }
22 );
23 for_each(begin(v), pos, print); // 输出分组后的数据
24
25 // min/max
26 auto value = std::minmax_element( //找出第一名和倒数第一
27     cbegin(v), cend(v)
28 );

```

在使用这些排序算法时，还要注意一点，它们对迭代器要求比较高，通常都是随机访问迭代器（minmax\_element 除外），所以**最好在顺序容器 array/vector 上调用**。

如果是 list 容器，应该调用成员函数 sort()，它对链表结构做了特别的优化。有序容器 set/map 本身就已经排好序了，直接对迭代器做运算就可以得到结果。而对无序容器，则不要调用排序算法，原因你应该不难想到（散列表结构的特殊性质，导致迭代器不满足要求、元素无法交换位置）。

## 查找算法

排序算法的目标是让元素有序，这样就可以快速查找，节约时间。



算法 `binary_search`，顾名思义，就是在已经排好序的区间里执行二分查找。但糟糕的是，它只返回一个 `bool` 值，告知元素是否存在，而更多的时候，我们是想定位到那个元素，所以 `binary_search` 几乎没什么用。

 复制代码

```
1 vector<int> v = {3,5,1,7,10,99,42}; // vector容器
2 std::sort(begin(v), end(v));        // 快速排序
3
4 auto found = binary_search(          // 二分查找，只能确定元素在不在
5     cbegin(v), cend(v), 7
6 );
```


想要在已序容器上执行二分查找，要用到一个名字比较怪的算法：`lower_bound`，它返回第一个“**大于或等于**”值的位置：

 复制代码

```
1 decltype(cend(v)) pos;                // 声明一个迭代器，使用decltype
2
3 pos = std::lower_bound(                // 找到第一个>=7的位置
4     cbegin(v), cend(v), 7
5 );
6 found = (pos != cend(v)) && (*pos == 7); // 可能找不到，所以必须要判断
7 assert(found);                        // 7在容器里
8
9 pos = std::lower_bound(                // 找到第一个>=9的位置
10    cbegin(v), cend(v), 9
11 );
12 found = (pos != cend(v)) && (*pos == 9); // 可能找不到，所以必须要判断
13 assert(!found);                      // 9不在容器里
```

`lower_bound` 的返回值是一个迭代器，所以就要做一点判断工作，才能知道是否真的找到了。判断的条件有两个，一个是迭代器是否有效，另一个是迭代器的值是不是要找的值。

注意 `lower_bound` 的查找条件是“**大于等于**”，而不是“等于”，所以它的真正含义是“大于等于值的第一个位置”。相应的也就有“大于等于值的最后一个位置”，算法叫 `upper_bound`，返回的是第一个“**大于**”值的元素。

 复制代码

```
1 pos = std::upper_bound(                // 找到第一个>9的位置
2     cbegin(v), cend(v), 9
```

```
3 );
```

因为这两个算法不是简单的判断相等，作用有点“绕”，不太好掌握，我来给你解释一下。

它俩的返回值构成一个区间，这个区间往前就是所有比被查找值小的元素，往后就是所有比被查找值大的元素，可以写成一个简单的不等式：

```
1 begin < x <= lower_bound < upper_bound < end
```


 复制代码

比如，在刚才的这个例子里，对数字 9 执行 `lower_bound` 和 `upper_bound`，就会返回 `[10,10]` 这样的区间。

对于有序容器 `set/map`，就不需要调用这三个算法了，它们有等价的成员函数 `find/lower_bound/upper_bound`，效果是一样的。

不过，你要注意 `find` 与 `binary_search` 不同，它的返回值不是 `bool` 而是迭代器，可以参考下面的示例代码：


```
1 multiset<int> s = {3,5,1,7,7,7,10,99,42}; // multiset, 允许重复
2
3 auto pos = s.find(7); // 二分查找，返回迭代器
4 assert(pos != s.end()); // 与end()比较才能知道是否找到
5
6 auto lower_pos = s.lower_bound(7); // 获取区间的左端点
7 auto upper_pos = s.upper_bound(7); // 获取区间的右端点
8
9 for_each( // for_each算法
10     lower_pos, upper_pos, print // 输出7,7,7
11 );
```

 复制代码

除了 `binary_search`、`lower_bound` 和 `upper_bound`，标准库里还有一些查找算法可以用于未排序的容器，虽然肯定没有排序后的二分查找速度快，但也正因为不需要排序，所以适应范围更广。

这些算法以 `find` 和 `search` 命名，不过可能是当时制定标准时的疏忽，名称有点混乱，其中用于查找区间的 `find_first_of/find_end`，或许更应该叫作 `search_first/search_last`。

这几个算法调用形式都是差不多的，用起来也很简单：

 复制代码

```
1 vector<int> v = {1,9,11,3,5,7}; // vector容器
2
3 decltype(v.end()) pos;          // 声明一个迭代器，使用decltype
4
5 pos = std::find(                 // 查找算法，找到第一个出现的位置
6     begin(v), end(v), 3
7 );
8 assert(pos != end(v));          // 与end()比较才能知道是否找到
9
10 pos = std::find_if(             // 查找算法，用lambda判断条件
11     begin(v), end(v),
12     [](auto x) {                // 定义一个lambda表达式
13         return x % 2 == 0;      // 判断是否偶数
14     }
15 );
16 assert(pos == end(v));          // 与end()比较才能知道是否找到
17
18 array<int, 2> arr = {3,5};      // array容器
19 pos = std::find_first_of(        // 查找一个子区间
20     begin(v), end(v),
21     begin(arr), end(arr)
22 );
23 assert(pos != end(v));          // 与end()比较才能知道是否找到
```

## 小结

C++ 里有上百个算法，我们不可能也没办法在一节课的时间里全部搞懂，所以我就精挑细选了一些我个人认为最有用的 `for_each`、排序和查找算法，把它们介绍给你。

在我看来，C++ 里的算法像是一个大宝库，非常值得你去发掘。比如类似 `memcpy` 的 `copy/move` 算法（搭配插入迭代器）、检查元素的 `all_of/any_of` 算法，用好了都可以替代很多手写 `for` 循环。

你可以课后仔细阅读 [标准文档](#)，对照自己的现有代码，看看哪些能用得上，再试着用算法来改写实现，体会一下算法的简洁和高效。

简单小结一下这次的内容：

1. 算法是专门操作容器的函数，是一种“智能 for 循环”，它的最佳搭档是 lambda 表达式；
2. 算法通过迭代器来间接操作容器，使用两个端点指定操作范围，迭代器决定了算法的能力；
3. `for_each` 算法是 `for` 的替代品，以函数式编程替代了面向过程编程；
4. 有多种排序算法，最基本的是 `sort`，但应该根据实际情况选择其他更合适的算法，避免浪费；
5. 在已序容器上可以执行二分查找，应该使用的算法是 `lower_bound`；
6. `list/set/map` 提供了等价的排序、查找函数，更适应自己的数据结构；
7. `find/search` 是通用的查找算法，效率不高，但不必排序也能使用。

和上节课一样，我再附送一个小技巧。

因为标准算法的名字实在是太普通、太常见了，所以建议你一定要显式写出“`std::`”名字空间限定，这样看起来更加醒目，也避免了无意的名字冲突。

## 课下作业

最后是课下作业时间，给你留两个思考题：

1. 你觉得 `for_each` 算法能完全替代 `for` 循环吗？
2. 试着自己总结归纳一下，这些排序和查找算法在实际开发中应该如何使用。

欢迎你在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎分享给你的朋友。我们下节课见。

# 课外小贴士

1. 容器通常还提供成员函数 `rbegin()`/`rend()`，用于逆序迭代（reverse），相应的也有全局函数 `rbegin()`/`rend()`、`crbegin()`/`crend()`。
2. `for_each` 算法还有一个不同于 `for` 循环的地方，它可以返回传入的函数对象。但因为 `lambda` 表达式只能被调用，没有状态，所以搭配 `lambda` 表达式的时候，`for_each` 算法的返回值就没有意义。
3. `equal_range` 算法可以一次性获得 `[lower_bound, upper_bound]` 这个区间，节约一次函数调用。
4. C++17 允许算法并行处理，需要传递 `std::execution::par` 等策略参数，提升运行效率。
5. C++20 引入了 `range` 的概念，在名字空间 `std::ranges` 提供基于范围操作的算法，可以不必显式写出 `begin()`、`end()`。而且它还重载了“|”，实现简洁的“管道”操作。

更多课程推荐

# MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇  
前阿里资深技术专家



涨价倒计时 🕒

今日秒杀 **¥79**，6月13日涨价至 **¥129**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 三分天下的容器：恰当选择，事半功倍

下一篇 14 | 十面埋伏的并发：多线程真的很难吗？

## 精选留言 (10)

写留言



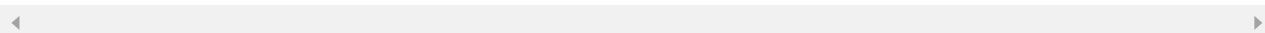
EncodedStar

2020-06-04

这个标题成功吸引了我。

展开 ∨

作者回复: 希望大家都能多用算法，少写for。



3



EncodedStar

2020-06-04

用c++很多年了，确实会遇到手写类似标准库的函数，手写完之后才发现标准库有同样功

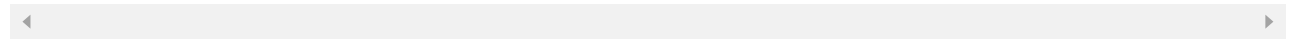


能高效的函数，简直让人感觉是闭门造车，哭笑不得。

既然老师文章都提到以后尽量用for\_each，我也觉得就可以替代for，以后尽量用老师教的，还有就是小技巧很实用！

展开 ▾

作者回复: 有时间多看看标准库文档，还有参考书，就可以少造些轮子，让自己也轻松一点。



💬 1

👍 2

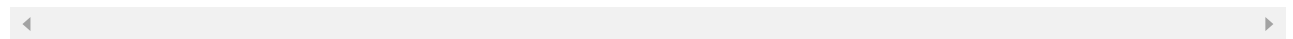


**Zivon**

2020-06-04

for\_each是不是无法返回pos，在需要得到元素位置的情况不太合适？

作者回复: 可以利用lambda表达式的闭包特性，用[&var]传出值。



💬

👍 1



**Geek\_5dc295**

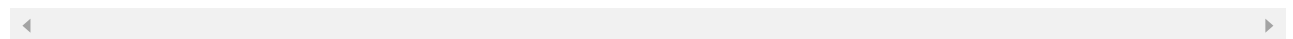
2020-06-06

有时候find 如果未找到对应位置不是会返回值有时候和npos对比判断，想问一下npos和迭代器之间是什么关系呀

展开 ▾

作者回复: npos应该是字符串string的未找到标志吧，我觉得这个应该算是设计字符串与容器时的一个失误，导致与迭代器的概念不兼容。

你可以把npos只理解成字符串位置相关的概念，表示未找到，不要把它和容器、迭代器联系起来，否则容易搞糊涂。



💬

👍



**范闲**

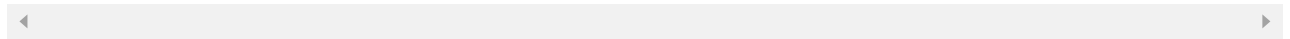
2020-06-06

1.用的比较多的是auto

2.for each range for看起来更像Python的语法糖，提高编程效率

展开 ▾

作者回复: 在算法里auto用的不多，多的是begin、end获取迭代器，再用lambda表达式处理元素，和for的差距还是挺大的。



**TC128**

2020-06-05

老师好，请问为什么有些算法可以传入数组地址，有些却不可以？比如：

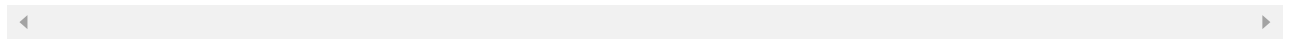
```
int myints[] = { 10, 20, 30, 40 };  
int * p;  
p = std::find (myints, myints+4, 30);
```

但for\_each就不可以。

展开 ▾

作者回复: 肯定是可以的，因为数组地址，也就是指针，它的作用和迭代器是一样的，泛型算法不会区别对待指针和迭代器。

可以把出错的代码贴出来看看，应该是用的有问题。

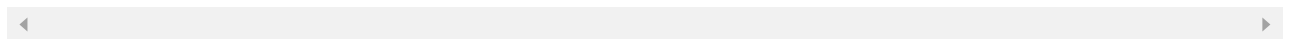


**robonix**

2020-06-05

老师，如果将二分查找算法应用在普通类元素上，是不是还得手写比较函数？

作者回复: 当然了，没有比较语义是肯定无法查找的。



**Confidant.**

2020-06-04

向问老师一个和今天内容没有关系的问题

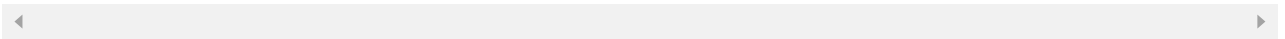
```
#include <iostream>  
#include <filesystem>  
...
```

展开 ▾

作者回复: filesystem是C++17里的吧，11/14里没有，我还没有用过，只是用过它的前身boost.filesystem。

从我的经验来看，path只是一个路径的字符串表示，不和实际的磁盘关联，就是一个普通对象，用shared\_ptr完全可以管理。

等我有机会用新的GCC来试试吧。



💬 1

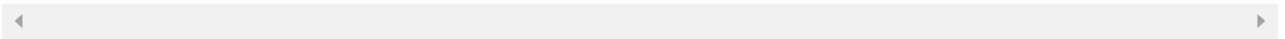


hy

2020-06-04

lambda表达式里面发生错误或者是出现异常外面是不是无法捕获的呀???

作者回复: 不会的，lambda就像是一个函数，发生异常自然会向外传，可以自己写代码试验一下。



Luca

2020-06-04

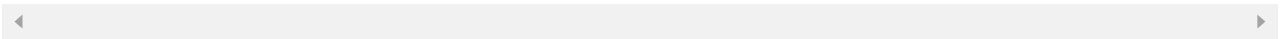
回答一下第一个问题，不知是否正确：for\_each循环不能完全代替for循环，比如在for循环中可以使用break跳出，而在for\_each中在语法层面是没有跳出的，如果要跳出的话，可能需要借助异常机制了。

当然，应用for\_each的函数式设计思想，也不应出现需要跳出循环的情况。

请老师与大家指正！

展开 ▾

作者回复: 在lambda表达式里可以用return，这样就可以结束循环，实现类似break的效果。



💬 2

